Trabalho Prático - ED1

Grupo 9:

Eike Cangussú, Matheus Alves,
Gabriel Meireles, Guilherme dos Anjos

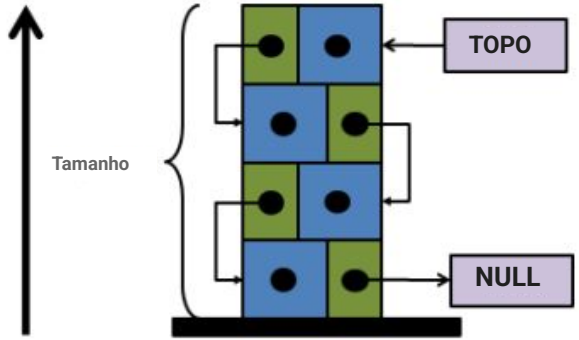
Implementação do TAD Pilha Genérico

- Last-in, First-out
- Necessidade de referência em cada nó
- Facilidade de implementação
- Complexidade das operações necessárias

- Estruturas:

```
typedef struct no{
    void* ch;
    struct no *prox;
}No;

typedef struct pilha{
    int tamanho;
    No *topo;
}Pilha;
```



- Implementação Genérica
- Funções Principais
- Funções Extras

- Vantagem de implementação genérica:
  - Reaproveitamento de código

```
Posicao *inicio
empilha(p, inicio);
Posicao *atual = (Posicao *)desempilha(p);
```

```
char *elemento
empilha(p, elemento);
char *topo = (char *)desempilha(p);
```

- Funções usuais:
  - empilha()
  - desempilha()
  - vazia()
  - esvazia()
  - tamanho()
- Funções extras:
  - destroiPilha()
  - mostrarTopo()

Solução dos Problemas

Problema 1:

Implementar um algoritmo que faça validação de expressões matemáticas digitadas pelo usuário. Utilizando a implementação do TAD Pilha para tipo genérico de dados. As expressões devem conter literais de A a J, operadores (+ (adição), - (subtração), / (divisão),* (multiplicação), ^ (potenciação)) e delimitadores de escopo (“(”, “)”, “[”, “]”, “{”, “}”).

Para resolver esse problema utilizamos duas funções:

validaExpTipoA();

validaExpTipoB();

validaExpTipoA();

```c
int validaExpTipoA(char *exp){
    Pilha *p = iniciaPilha();
    int valido = 1; // variavel flag de validação é iniciada como verdadeiro

    for(int i = 0; exp[i] != '\0'; i++){ // percorre cada caractere até chegar no fim '\0'
        char c = exp[i];
        if(c == '(' || c == '[' || c == '{'){  // verifica se o caratere é um delimitador de abertura
            char *elemento = (char*)malloc(sizeof(char));
            *elemento = c; // armazena o caractere
            empilha(p, elemento); // adiciona o caractere na pilha
        }

        else if( c == ')' || c == ']' || c == '}'){ // verifica se é um delimitador de fechamento
            if(vazia(p)){ // verifica se pilha está vazia ou seja se não existe abertura
                valido = 0; // se não existir abertura correspondente na pilha, marca a flag como invalida
                break;
            }

            char *topo = (char *)desempilha(p); // retira da pilha o ultimo delimitador e salva na variavel para fazer uma comparação
            if((c == ')' && *topo != '(') ||  // verifica se esse o delimitador de fechamento é diferente do de abertura
               (c == ']' && *topo != '[') ||
               (c == '}' && *topo != '{')) {
                valido = 0; // se for diferente, marca a flag como invalido
                free(topo); // libera memoria
                break;
            }

            free(topo); // libera memoria mesmo se não sastisfazer o condicional acima
        }
    }
```

validaExpTipoA();

...

```
    if(valido && !vazia(p)){ // verifica se após percorrer a expressão até o final ainda existe elementos na pilha
        valido = 0; // se ainda houver, significa que existem delimitadores em aberto na expressão marcando a flag como invalida
    }

    esvazia(p); //esvazia a pilha se ainda houverem elementos

    destroiPilha(p); // destroi a estrutura da pilha e limpa a memoria

    return valido; // retorno de validacao ou não da expressão, com 0 para invalido ou 1 para valido
}
```

validaExpTipoB();

```
int validaExpTipoB(char * exp){
    Pilha *p = iniciaPilha();
    int valido = 1;  // variavel flag de validação é iniciada como verdadeiro

    for(int i = 0; exp[i] != '\0'; i++){ // percorre cada caractere até chegar no fim '\0'
        char c = exp[i];
        if(c == '(' || c == '[' || c == '{' ){ // verifica se o caratere é um delimitador de abertura
            if (!vazia(p)) { // verifica se a pilha nao esta vazia
                char* top_char = (char*)mostrarTopo(p); // vai obter o elemento do topo sem desempilhar

                if (c == '[') { // verifica se esta abrindo colchete e se o topo é parenteses
                    if (*top_char == '(') {
                        valido = 0; // marca flag como invalido
                        break;
                    }
                } else if (c == '{') { // verifica se esta abrindo chave e se o topo é colchete ou parenteses
                    if (*top_char == '(' || *top_char == '[') { // chave dentro de parenteses ou chave dentro de colchete
                        valido = 0; // marca flag como invalido
                        break;
                    }
                }
            }

            char *elemento = (char*)malloc(sizeof(char)); // se passou nas verificações, o delimitador é alocado e empilhado
            *elemento = c;
            empilha(p, elemento);

        }

        else if(c == ')' || c == ']' || c == '}'){ // verifica se é um delimitador de fechamento
            if(vazia(p)){  // verifica se pilha está vazia ou seja se não existe abertura
                valido = 0;  // se não existir abertura correspondente na pilha, marca a flag como invalida
                break;
            }
```

validaExpTipoB();

...

```
            char *topo = (char *)desempilha(p);  // retira da pilha o ultimo delimitador e salva na variavel para fazer uma comparação
            if ((c == ')' && *topo != '(') ||  // verifica se esse o delimitador de fechamento é diferente do de abertura
                (c == ']' && *topo != '[') ||
                (c == '}' && *topo != '{')) {
                valido = 0;  // se for diferente, marca a flag como invalido
                free(topo);  // libera memoria
                break;
            }

            free(topo);  // libera memoria mesmo se não sastisfazer o condicional acima
        }

    }

    if(valido && !vazia(p)){  // verifica se após percorrer a expressão até o final ainda existe elementos na pilha
        valido = 0;  // se ainda houver, significa que existem delimitadores em aberto na expressão marcando a flag como invalida
    }

    esvazia(p); //esvazia a pilha se ainda houverem elementos

    destroiPilha(p);  //destroi a estrutura da pilha e limpa a memoria

    return valido;  // retorno de validacao ou não da expressão, com 0 para invalido ou 1 para valido
}
```

Problema 2:

Implementar um programa para manipulação de expressões matemáticas envolvendo variáveis literais de A a J, operadores (+ (adição), - (subtração), / (divisão),* (multiplicação),^ (potenciação)) e os delimitadores de escopo tipo parênteses (“(”, “)”). Utilize a implementação do TAD Pilha para tipo genérico de dado solicitada no Objetivo do Trabalho.Para tal, o programa deve ter as seguintes funcionalidades:

Funções principais

```
converteInfixa(char *infixa)
precedencia(char operador)
operador(char exp)
entradavalida( char exp)
expressaoValida(char *infixa)
```

bool expressaoValida(char *infixa)

```c
bool expressaoValida(char *infixa) {
    /*
        Verifica se a expressão infixa é válida
        Retorna true se for válida, false caso contrário
    */
    int parenCount = 0;
    for (int i = 0; infixa[i] != '\0'; i++) {
        if (infixa[i] == '(') {
            parenCount++;
        } else if (infixa[i] == ')') {
            parenCount--;
            if (parenCount < 0) return false; // Parênteses fechados sem correspondência
        }
    }

    return parenCount == 0; // Todos os parênteses devem estar balanceados
}
```

bool entradavalida(char exp)

```
bool entradavalida( char exp){
    /*
        Verifica se o caracter é valido
        Caracteres validos são variáveis literais de A a J,
        operadores (+ (adição), - (subtração), / (divisão),* (multiplicação),^ (potenciação))
        e os delimitadores de escopo tipo parênteses ( “(”, “)”)
    */
```

bool operador(char exp)

```
bool operador(char exp){
    /*
        Verifica se o caracter é um operador
    */
    if (exp == '+' || exp == '-' || exp == '/' || exp == '*' || exp == '^')
    {
        return true;
    }
    return false;
}
```

int precedencia(char operador)

```c
int precedencia(char operador) {
    /*
        Retorna a precedência do operador
        Maior valor significa maior precedência
    */
    switch (operador) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0; // Para operadores não reconhecidos
    }
}
```

void converteInfixa(char *infixa)

```
void converteInfixa(char *infixa)
{
    /*
        Converte uma expressão infixa para posfixa
        A expressão infixa é passada como parâmetro
        A expressão posfixa é impressa na tela
    */
```

void converteInfixa(char *infixa)

```
for (int i = 0; infixa[i] != '\0'; i++) {
    char c = infixa[i];

    if (entradavalida(c)) {

        if (c >= 'A' && c <= 'J') { // Se for uma variável literal
            posfixa[j++] = c;
        }
        else if (c == '(') { // Se for um parêntese de abertura
            char *elemento = (char*)malloc(sizeof(char));
            *elemento = c;
            empilha(pilha, elemento);
        }
        if (operador(c)) {
```

void converteInfixa(char *infixa)

```
// Desempilhar os operadores restantes na pilha
while (!vazia(pilha)) {
    char *topoElem = (char *)desempilha(pilha);
    posfixa[j++] = *topoElem;
    free(topoElem);
}

//print posfixa
printf("\nPosfixa: %.*s\n", j, posfixa);
posfixa[j] = '\0'; // Finaliza a string posfixa
destroiPilha(pilha);
```

int main()

```c
int main() {
    /*
        Função principal que executa o programa
        Solicita ao usuário a expressão a ser convertida
        e chama a função de conversão
    */
    char infixa[100];
    int escolha = 0; // Inicializa a variável escolha

    // MENU DE OPÇÕES

        printf("Escolha o formato da expressao a ser inserida:\n");
        printf("1. Forma Posfixa\n");
        printf("2. Forma Infixa\n");
        printf("Digite sua escolha: ");
        scanf("%d", &escolha);
```

Problema 3:

Dada uma planta de uma casa, sua tarefa é contar o número de cômodos que ela possui. A planta é representada por uma matriz de n × m quadrados, e cada quadrado é ou um piso ou uma parede. Você pode se mover para a esquerda, direita, para cima e para baixo através
dos quadrados de piso. Utilize a implementação do TAD Pilha para tipo genérico de
dados.

Para resolver esse problema utilizamos duas funções :

buscaComodo (n , m , i , j)

main ( )

Estruturas e Variáveis Auxiliares

```c
#include "pilha.h"
#include <string.h>
#define MAX 1000        /* Define o tamanho maximo da matriz*/
char mapa[MAX][MAX];    /* Variavel para o mapa */
int visitou[MAX][MAX]; /* Variavel para saber qual parte do mapa visitou (se sim = 1, se nao = 0) */

typedef struct posicao /* Estrutura para guardar uma posicao no mapa ( sendo x = linhas e y = colunas)*/
{
    int x, y;
} Posicao;

int dy[4] = {-1, 1, 0, 0}; /* Direcao para cima e para baixo */
int dx[4] = {0, 0, -1, 1}; /* Direcao para esquerda e para direita */
```

buscaComodo(n , m ,i , j)

```
Posicao *inicio = (Posicao *)malloc(sizeof(Posicao)); /* aloca memoria para a posicao inicial */
if (inicio == NULL)                                   /*verifica se conseguiu alocar a memoria*/
{
    printf("ERRO de Alocação!");
    exit(1);
}
/*define as posicoes inicais*/
inicio->x = i;
inicio->y = j;
empilha(p, inicio); /*coloca a posical inicial na pilha*/
visitou[i][j] = 1;  /*marca a posicao inicial como visitada*/
while (!vazia(p))   /*enquanto a pilha nao estiver vazia fica rodando*/
{
    Posicao *atual = desempilha(p); /*retira o elemento do topo da pilha*/
    for (int d = 0; d < 4; d++)     /*percorrer as 4 direcoes*/
    {
        int ni = atual->x + dy[d];                /*pula para a proxima linha*/
        int mj = atual->y + dx[d];                /*pula para a proxima coluna*/
```

Continuação da buscaComodo

```
            if (ni >= 0 && ni < n && mj >= 0 && mj < m) /*verifica se está dentro do limite do mapa*/
            {
                if (!visitou[ni][mj] && mapa[ni][mj] == '.') /*verifica se ainda nao visitou e é um piso*/
                {
                    visitou[ni][mj] = 1;                     /*marca como visitado*/
                    Posicao *nova = malloc(sizeof(Posicao)); /*cria uma nova posicao para empilhar*/
                    if (nova == NULL)                        /*verifica a alocacao de memoria */
                    {
                        printf("ERRO de Alocação!");
                        exit(1);
                    }
                    nova->x = ni;
                    nova->y = mj;
                    empilha(p, nova); /*empilha a nova posicao*/
                }
            }
        }
        free(atual); /*libera a memoria da posicao atual*/
    }
    destroiPilha(p); /*libera toda a memoria usada pela pilha*/
}
```

main()

```c
int main()
{
    int comodos = 0; /*contador de comodos*/
    char linha[MAX]; /*buffer para ler as linhas do terminal*/
    /*Lê as dimensoes do mapa */
    int n, m; /*variaveis*/
    if (scanf("%d %d", &n, &m) != 2)
    {
        printf("Erro na leitura de n e m.\n");
        return 1;
    }

    while (getchar() != '\n')
        ;                                       /* limpa o buffer do teclado*/
    if (n <= 0 || m <= 0 || n > MAX || m > MAX) /*verifica os valores de n , m*/
    {
        printf("Dimensoes invalidas! Informe valores entre 1 e %d.\n", MAX);
        return 1;
    }
```

Continuação da main()

```c
for (int i = 0; i < n; i++)
{
    if (fgets(linha, sizeof(linha), stdin) == NULL) /*verifica a leitura da linha*/
    {
        printf("Erro na leitura do mapa.\n");
        return 1;
    }
    linha[strcspn(linha, "\n")] = '\0'; /* remove o \n se houver */
    for (int j = 0; j < m; j++)
    {
        mapa[i][j] = linha[j];
        visitou[i][j] = 0; /* marca como não visitada */
    }
}

/*Busca por comodos*/
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        if (mapa[i][j] == '.' && !visitou[i][j]) /* verifica se encontrou um piso nao visitado*/
        {
            buscaComodo(n, m, i, j); /*marca todo o comodo*/
            comodos++;               /*incrementa a contagem de comodos*/
        }
    }
}

printf("%d\n", comodos); /* escreve no terminal a quantidade de cômodos*/
```